May 1989

Deriving Mixed Evaluation From Standard Evaluation for a Simple Functional Language

John Hannan
University of Pennsylvania

Dale Miller
University of Pennsylvania

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-89-28.

Deriving Mixed Evaluation From Standard Evaluation for a Simple Functional Language

Abstract

We demonstrate how a specification for the standard evaluation of a simple functional programming language can be systematically extended to a specification for mixed evaluation. Using techniques inspired by natural semantics we specify a standard evaluator by a set of inference rules. The evaluation of programs is then performed by a restricted kind of theorem proving in this logic. We then describe a systematic method for extending the proof system for standard evaluation to a new proof system that provides greater flexibility in treating bound variables in the object-level functional programs. We demonstrate how this extended proof system provides the capabilities of a mixed evaluator and how correctness with respect to standard evaluation can be proved in a simple and direct manner. The current work focuses only on a primitive notion of mixed evaluation for a simple functional programming language, but we believe that our methods will extend to more sophisticated kinds of evaluations and richer languages.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-89-28.

Deriving Mixed Evaluation From Standard Evaluation For A Simple Functional Language

MS-CIS-89-28
LINC LAB 150

John Hannan
Dale Miller

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104-6389

May 1989

This is a revised version os a paper appearing at the International Conference on the Mathematics of Program Construction, Twente University, The Netherlands, June 1989.

Acknowledgements:
The first author is supported in part by a fellowship from the Corporate Research and Architecture Group, Digital Equipment Corporation, Maynard, MA USA. Both authors are supported in part by NSF grants CCR-05596-MCS-8219196-CER, IRI84-10413-A02, ONR N00014-88-K-0633, DARPA grant N00014-85-K-0018 and U.S. Army grants DAA29-84-K-0061, DAA29-84-9-0027

Deriving Mixed Evaluation from Standard Evaluation for a Simple Functional Language[1]

JOHN HANNAN and DALE MILLER

Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389 USA

We demonstrate how a specification for the standard evaluation of a simple functional programming language can be systematically extended to a specification for mixed evaluation. Using techniques inspired by natural semantics we specify a standard evaluator by a set of inference rules. The evaluation of programs is then performed by a restricted kind of theorem proving in this logic. We then describe a systematic method for extending the proof system for standard evaluation to a new proof system that provides greater flexibility in treating bound variables in the object-level functional programs. We demonstrate how this extended proof system provides the capabilities of a mixed evaluator and how correctness with respect to standard evaluation can be proved in a simple and direct manner. The current work focuses only on a primitive notion of mixed evaluation for a simple functional programming language, but we believe that our methods will extend to more sophisticated kinds of evaluations and richer languages.

1 Introduction

The formal derivation and correctness of program analysis tools play a central role in many programming language research efforts. In this paper we focus on evaluators for programming languages. We shall use natural deduction techniques to specify and derive evaluators for a simple functional language. With a natural deduction theorem prover, one constructs formal proofs of propositions using a particular set of inference rules. If we encode programs as terms then we can build propositions expressing relationships between programs. A proof system can then axiomatize such relationships. This approach to program analysis shares much with the work on structural operational semantics [19] and natural semantics [12].

Since what we shall call "mixed evaluation" is related to the terms "partial evaluation" and "mixed

[1]This is a revised version of a paper appearing at the International Conference on the Mathematics of Program Construction, Twente University, The Netherlands, June 1989.

computation," we provide a brief description of our use of these terms. *Standard evaluation* refers to a conventional notion of evaluation or interpretation of functional programs. *Partial evaluation* is the process of constructing a new program given some original program and a part of its input [3]. In general terms, it can be described as follows. Let f be some functional program of two arguments x and y and consider the application $f(c, y)$ for some constant (known) value c and variable (unknown) value y. We wish to construct a new functional program f_c such that $f_c(y) = f(c, y)$ for all values of y, such that for any value of y, computing $f_c(y)$ should be easier (or faster) than computing $f(c, y)$. Such improvement is possible by "compiling" the information that $x = c$ in f into the definition of f_c.

Mixed evaluation, also called *symbolic evaluation* in [5], is the more general process of evaluating expressions, which may contain free variables (i.e., not bound to any values), to some canonical form. The key task of this process is properly treating the interaction between known and unknown (symbolic) values. This combination of known and unknown values suggests the adjective "mixed." While standard evaluation typically interprets variables by providing a mapping between the variables and their associated values, mixed evaluation requires some method for interpreting variables as objects themselves.

The importance of mixed evaluation was elucidated by Futamura [3] when he described the construction of compiled programs, compilers, and compiler generators via mixed evaluation. Thus mixed evaluation is a means for understanding and constructing a wide range of translation tools. *But where do mixed evaluators come from?* In particular, can mixed evaluators be formally derived from standard evaluators? Few research efforts have addressed the formal construction of mixed evaluators using principled techniques. We address this question by demonstrating how, for a simple functional programming language, a specification for mixed evaluation can be derived from a specification for standard evaluation.

The remainder of this paper is organized as follows. In Section 2 we introduce a simple functional programming language, giving both a concrete and an abstract syntax. Following this we specify a standard evaluator for this language in Section 3. In Section 4 we use the signature of the functional program's abstract syntax to construct a mixed evaluator and in Section 5 we prove a form of its correctness. Issues of implementation are discussed in Section 6. Summary comments and a description of related work are provided in Section 7.

2 Abstract Syntax as Lambda Terms

We introduce a simple functional programming language for which we shall specify two evaluators in subsequent sections. There is strong connection between the choice of *abstract syntax* for the formal representation of functional programs and the complexity of the presentation of these evaluators. An appropriate choice of abstract syntax will later facilitate our specification of simple and declarative evaluators. It is this simple and declarative aspects of one evaluator (for standard evaluation) that will permit it to be simply and automatically enhanced to yield another evaluator (for mixed evaluation).

We distinguish between *concrete syntax*, which provides a convenient human-readable presentation for programs, and abstract syntax. Let E be the functional language whose concrete syntax is defined by the following grammar:

$$E \ ::= \ \mathbf{C} \ \mid \ \mathbf{x} \ \mid \ \mathbf{if}\ E\ \mathbf{then}\ E\ \mathbf{else}\ E \ \mid \ (E\,E) \ \mid$$

$C : tm$	$lamb : (tm \to tm) \to tm$
$if : tm \to tm \to tm \to tm$	$let : (tm \to tm) \to tm \to tm$
$@ : tm \to tm \to tm$	$fix : (tm \to tm) \to tm$

FIGURE 1
Typed Constants for E's Abstract Syntax

$$\mathbf{\lambda x}.E \quad | \quad \mathbf{let\ x} = E \ \mathbf{in}\ E \quad | \quad \mathbf{fix\ x}.E$$

The symbol **C** ranges over primitive constants including the integers and booleans.

We now define an abstract syntax for the programs of E. We shall view an evaluator as a program that manipulates terms denoting E-programs. Thus we must define the set of terms and a method for encoding E-programs into such terms. We shall use simply typed λ-terms as the representation language. To define our abstract syntax for E we begin by introducing the base type tm and a set of typed constants that we shall use to construct terms denoting E-programs. (See Figure 1.) Notice that the constants *lamb*, *let* and *fix* are second-order, that is, they each require a functional argument of type $tm \to tm$. In the examples that follow, M will be used as a second-order variable of this meta-type and e_i and α_i will be meta-variables of type tm.

Using the new constants of Figure 1 we can build up λ-terms forming an abstract syntax for E as follows. For constants and variables in the concrete syntax we introduce associated constants and variables of type tm to the abstract syntax. For the **if** statement we introduce the new constant *if* such that if the three terms e_1, e_2, e_3 are of type tm, then $(if\ e_1\ e_2\ e_3)$ is a term of type tm. Application is made explicit with the infix operator '@' so that $e_1@e_2$ represents the expression denoted by the term e_1 applied to e_2. For lambda abstraction we introduce the constructor *lamb* that takes a meta-level abstraction of the form $\lambda x.e$, in which x and e are of meta-type tm, and produces a term of type tm. For example, the concrete syntax for lambda abstraction is $\mathbf{\lambda x}.E$ while its abstract syntax form is $(lamb\ \lambda x.e)$ (in which e is the abstract syntax form of E). Similar to *lamb*, the *let* construct uses a meta-term M of the form $\lambda x.e$ to represent the binding of an identifier. Thus the concrete syntax $\mathbf{let\ x} = E_1\ \mathbf{in}\ E_2$ is given by the abstract term $(let\ \lambda x.e_2\ e_1)$ in which e_1 and e_2 are the abstract syntax forms of E_1 and E_2, respectively. To represent the recursive **fix** construct we introduce the *fix* constant, which again uses an explicit abstraction to capture the binding. An example of this construction is given below.

The language E shall also contain several constants denoting lists and primitive operations on list. That is, we shall assume that E also contains the constants $cons, nil, car, cdr$, and *null* all at primitive type tm. Given this typing scheme, to apply the constant *null* to an argument, say e, we must write $(null@e)$. It is possible to provide *null* with the type $tm \to tm$ and to write this application as simply $(null\ tm)$. For most aspects of this paper, this choice of typing for these primitive constants will not make a significant difference. We shall, however, adopt the convention that those functions that do not correspond to special forms in ML will be denoted with the simple type tm.

We will not discuss any primitive operations on integers or booleans in this paper. They are, of course, important to have in the full language but including them here is neither difficult nor

illuminating. In the following and subsequent examples, we systematically drop the apply "@" operator in order to make examples more readable. Consider the following expression that defines the append function and then applies it to two lists.

let app = (fix f.λk.λl.(if (empty k) then l else (cons (hd k) (f (tl k) l))))
in (app [1] [2]).

The corresponding term in the abstract syntax is

(let λapp (app (cons 1 nil) (cons 2 nil))
(fix λf(lamb λk(lamb λl(if (empty k) l (cons (hd k) (f (tl k) l))))))).

Note how the four bindings in the concrete syntax (**app**, **f**, **k**, **l**) are translated into explicit λ-abstractions in the abstract syntax.

We shall assume that the reader is familiar with the notions of β-conversion and β-normal form for simply typed λ-terms. For discussions on the motivation and advantages of using simply typed λ-terms to encode functional programs, see [7, 8, 11, 13, 18].

3 Standard Evaluation

We now present an evaluator for E, which we shall call the standard evaluator for E to distinguish it from a second evaluator defined later. We shall divide the description of this evaluator into two parts. The *declarative* aspects of it shall be presented using a proof system similar in style to structural operational semantics and natural semantics [12, 19]. Computing will be equated to finding proofs in this proof system. The *control* aspects of this evaluator, that is, the search strategy used to find proofs, shall be particularly simple for the standard evaluator. Control of our second evaluator will be much more difficult. Logic programming provides a good setting for relating these different aspects of evaluators. A particularly relevant logic programming language is discussed in Section 6.

To represent the proposition that a given program evaluates to a particular value, we need to add to our term language a new type o for proposition, and a binary, infix constant $\longrightarrow$ of type $tm \rightarrow tm \rightarrow o$. The basic propositions for both evaluators we consider will, therefore, be of the form $e \longrightarrow e'$ where e and e' are both closed λ-terms denoting expressions of E. If this proposition is provable then we shall say that e evaluates to e'. Since evaluation is represented as a relation, a given program may "evaluate" to more than one value. While this is not true of the standard evaluator (see Proposition 3.1), it will be true of our second evaluator.

The declarative aspects of the standard evaluator are given by inference rules provided in Figure 2. Proofs using these rules are defined in the usual, natural deduction fashion with only the following difference. Whenever an inference rule involves a formula of the form $(M\ e)$ where M is a term of type $tm \rightarrow tm$ and of the form $\lambda x\ t$, then we shall assume that this non-β-normal term is an abbreviation for the term that results from substituting e for the free occurrences of variable x in the term t. Notice that if M and e are β-normal terms of E, then the result of this substitution is again a β-normal term of E. That is, no new redexes are introduced by substitution. For this reason, we shall generally limit ourselves to considering only proofs in which all terms from E are in β-normal form. The non-

$$c \longrightarrow c \tag{1}$$

$$\frac{e_1 \longrightarrow true \qquad e_2 \longrightarrow \alpha}{(if\ e_1\ e_2\ e_3) \longrightarrow \alpha} \qquad \frac{e_1 \longrightarrow false \qquad e_3 \longrightarrow \alpha}{(if\ e_1\ e_2\ e_3) \longrightarrow \alpha} \tag{2a, 2b}$$

$$(lamb\ M) \longrightarrow (lamb\ M) \tag{3}$$

$$\frac{e_1 \longrightarrow (lamb\ M) \qquad e_2 \longrightarrow \alpha_2 \qquad (M\ \alpha_2) \longrightarrow \alpha}{(e_1 @ e_2) \longrightarrow \alpha} \tag{4}$$

$$\frac{e_2 \longrightarrow \alpha_2 \qquad (M\ \alpha_2) \longrightarrow \alpha}{(let\ M\ e_2) \longrightarrow \alpha} \qquad \frac{(M\ (fix\ M)) \longrightarrow \alpha}{(fix\ M) \longrightarrow \alpha} \tag{5, 6}$$

FIGURE 2
Inference Rules for the Standard Evaluator for E

β-normal terms appearing in the inference rules for *lamb*, *fix*, and @ are intended as a shorthand for β-normal terms.

The first rule in Figure 2 specifies that the constants of E evaluate to themselves. The next two rules treat the *if* expression in a natural way: the conditional part, e_1, must evaluate to *true* or *false* for a proof to be found. Rule (3) states that object-level λ-abstractions also evaluate to themselves. In the rule for application (4), meta-level substitution correctly captures the notion of function application (with a call-by-value semantics). In terms of our encoding at the abstract syntax level, this rule simply states that @ o *lamb* is the identity function on terms of type $(tm \rightarrow tm)$. Similar comments apply to the rule for *let* (5). In the rule for recursion (6), the fixed point operator is given the obvious unfolding meaning. This again makes explicit use of substitution at the meta-level since the meta-term M is applied to the term $(fix\ M)$. The result of this substitution replaces recursive calls with the body of the recursive program, namely $(fix\ M)$. Static scoping is ensured with this specification because substitution, as a means of propagating binding information, guarantees that the abstracted identifiers are replaced with their associated value prior to evaluating abstractions. Thus we shall not need closures for manipulating abstractions.

Recall that we included in E some primitive constants for manipulating lists. The inference rules for specifying the behavior of the standard evaluator for these additional constants are given in Figure 3.

If the proposition $e \longrightarrow e'$ has a proof using only the inference rules in Figures 2 and 3 then we shall write $\vdash_{se} e \longrightarrow e'$ and refer to the resulting proof as an *se*-proof. Notice that any such proof does not use two structural aspects of general natural deduction proofs: arguments from hypotheses and critical variables (eigen-variables) . Our second interpreter, however, will make use of both of these aspects of natural deduction proofs.

Given some closed expression e, we can think of the evaluation of e as the process of finding a

$$nil \longrightarrow nil \qquad \frac{e_1 \longrightarrow \alpha_1 \qquad e_2 \longrightarrow \alpha_2}{(cons\ e_1\ e_2) \longrightarrow (cons\ \alpha_1\ \alpha_2)}$$

$$\frac{e \longrightarrow (cons\ \alpha_1\ \alpha_2)}{(car\ e) \longrightarrow \alpha_1} \qquad \frac{e \longrightarrow (cons\ \alpha_1\ \alpha_2)}{(cdr\ e) \longrightarrow \alpha_2}$$

$$\frac{e \longrightarrow nil}{(null\ e) \longrightarrow true} \qquad \frac{e \longrightarrow (cons\ \alpha_1\ \alpha_2)}{(null\ e) \longrightarrow false}$$

FIGURE 3
Additional Inference Rules for Some Primitive Constants

proof of the proposition $\vdash_{se} e \longrightarrow e'$, for some expression e'. It is easy to see from the inference rules, that the only terms e' for which a proposition of the form $e \longrightarrow e'$ has a proof are either integers, boolean constants, object-level lambda expressions, or nested lists of such objects. We shall refer to such terms as (*proper*) *values* and use α to denote such terms. Note that we have not supplied an explicit evaluation ordering to either the inference rules or the propositions in a premise. Thus one should think of nondeterministically searching for a proof via these inference rules. An actual implementation, however, must make some commitment.

The following proposition about the standard evaluator is easily proved.

PROPOSITION 3.1 Given a closed λ-term e of type tm, there is at most one proof of a proposition of the form $e \longrightarrow \alpha$, where α is some term.

PROOF. Let $\mathcal{P}(e)$ be the set of proofs of the proposition $e \longrightarrow \alpha$ for some α. Let $\#(e)$ be the height of the smallest tree in $\mathcal{P}(e)$. Let e be picked so that $\#(e)$ is minimal among those terms for which $\mathcal{P}(e)$ contains two or more members. The ordering among terms here is according to the number of non-logical constants (i.e., the constants from Figure 1) occurring in the term. The proof proceeds by considering the structure of e. We include here just two cases that illustrate the proof.

(i) Assume $e = (if\ e_1\ e_2\ e_3)$. Then proofs in $\mathcal{P}(e)$ have either

$$\frac{e_1 \longrightarrow true \qquad e_2 \longrightarrow \alpha}{(if\ e_1\ e_2\ e_3) \longrightarrow \alpha} \quad \text{or} \quad \frac{e_1 \longrightarrow false \qquad e_3 \longrightarrow \alpha}{(if\ e_1\ e_2\ e_3) \longrightarrow \alpha}$$

as their last inference rule. If all proofs in $\mathcal{P}(e)$ are of the first form then $\mathcal{P}(e_2)$ must have at least two different proofs; but since e_2 is a smaller term than e, this contradicts the selection of e. Similarly, if all proofs in $\mathcal{P}(e)$ are of the second form then $\mathcal{P}(e_3)$ must have at least two different proofs; but since e_3 is a smaller term than e, this contradicts the selection of e. Finally, we have the case in which $\mathcal{P}(e)$ contains proofs of both forms. But then $\mathcal{P}(e_1)$ has at least two different proofs, again contradicting the choice of e.

(*ii*) Assume $e = (\mathit{fix}\ M)$. Then all proofs in $\mathcal{P}(e)$ have the last inference rule

$$\frac{(M\ (\mathit{fix}\ M))\longrightarrow\alpha}{(\mathit{fix}\ M)\longrightarrow\alpha}.$$

Thus the cardinality of $\mathcal{P}(e)$ and $\mathcal{P}(M\ (\mathit{fix}\ M))$ must be the same. But $\#(e) = \#(M\ (\mathit{fix}\ M))+1$ which contradicts the selection of e.

The remaining cases are carried through similarly. □

We say that e *has a value* if there is a proof of $e\longrightarrow\alpha$ for some term α. By virtue of the above proposition, we may say that if e has a value, it has a unique value, i.e., the α such that $e\longrightarrow\alpha$ is provable. Notice that we shall not insist that all functional programs of E have values. An object-level, ML-style typing scheme could be used to remove certain programs that do not have values. Of course, other programs may fail to have values since they never terminate. We shall not identify such non-terminating programs with the "undefined value" as is often done in denotational semantics.

By virtue of Proposition 3.1, it is possible to use theorem proving or logic programming techniques to turn the proof system specification of an evaluator into an actual implementation. For example, all the inference rules for constructing *se*-proofs can be represented as Horn clauses over a language where first-order terms are replaced by simply typed λ-terms. For example, inference rule (4) could be written as the Horn clause

$$\forall e_1, e_2, \alpha, \alpha_2, M[e_1\longrightarrow(\mathit{lamb}\ M)\ \ \&\ \ e_2\longrightarrow\alpha_2\ \ \&\ \ (M\ \alpha_2)\longrightarrow\alpha\ \Rightarrow\ (e_1@e_2)\longrightarrow\alpha].$$

Here, conjunction is denoted by the logical constant & of type $o\rightarrow o\rightarrow o$. Determining that e has a value is equivalent to proving that the formula $\exists\alpha(e\longrightarrow\alpha)$ has a proof from Horn clauses that result from translating the inference rules of the evaluator. As is well known, if such proofs can be found, it can be assumed that they contain the witness for this existential, which would, of course, be the value of e. Because the rules for standard evaluation have a very regular structure, a very naive control strategy, such as the depth-first control of Prolog, would serve to find values whenever they exist. Thus, when we refer to standard evaluation as an actual, deterministic process, we shall think of it as this kind of Prolog-like execution.

Evaluation of this kind can be viewed as a kind of one-way rewriting. In this evaluator, however, rewriting always takes place at the top-most level of an expression. Rewriting could be attempted on proper subexpressions of a given expression, although this would be problematic if the subexpression was inside the scope of an object-level abstraction. Notice that this evaluator deals with object-level abstractions in one of two ways: it either treats it as essentially "quoted," as in the case of *lamb*, or it removes it by substituting a value in for it, as in the case of @, *let*, and *fix*.

Computing inside an abstraction has at least two problems. The first is that of correctly evaluating an expression containing an abstracted variable: the evaluator currently only deals with expressions whose top-level symbol is a constant declared in Section 2. A reasonable method for solving this problem is to specify that when an abstracted variable is encountered, it should evaluate to itself, that is, it should be treated as a "quoted" expression. The challenge here is to see how a proof system might support such a treatment of abstracted variables. The second problem of rewriting within an abstraction is that generally the result of such rewriting will not be a proper value. Consider, for example, the expression

$$(\mathit{lamb}\ \lambda x(\mathit{if}\ (\mathit{null}\ x)\ e_1\ e_2)).$$

Applying the standard evaluator to the expressions e_1 and e_2 would yield proper values if their evaluation was independent of any value assigned to x. So assume that e_1 and e_2 evaluate to the expressions α_1 and α_2, respectively. Further evaluation of the *if* expression is impossible as the value of $(null\ x)$ is dependent on having a known value for x. We can only reduce the above expression to

$$(lamb\ \lambda x(if\ (null\ x)\ \alpha_1\ \alpha_2)).$$

Thus, we must deal with "improper" or generalized values. As this example illustrates, an *if* statement can be rewritten in three different ways: Figure 2 provides two ways and the third way replaces an *if* statement with another *if* statment where its arguments may have been rewritten. This last observation reveals a cost in doing a more liberal rewriting of terms: evaluation may become more non-deterministic.

In the next section, we specify an extension to standard evaluation that is capable of systematically descending into abstractions and correctly handling abstracted variables.

4 Mixed Evaluation

The notion of descending into an abstraction to perform rewrites is often referred to as *mixed evaluation* since evaluation must be done not only on terms of E but also on terms containing abstracted variables and these are often treated as *symbolic* values. Thus, computations on "real" and symbolic values must be mixed together.

To obtain a *mixed evaluator* for E we first specify its proof system, which will be obtained by adding proof rules to those for the standard evaluator. As the discussion from the last section indicates, we should add the following inference figure to the standard evaluator.

$$\frac{e_1 \longrightarrow e_1' \qquad e_2 \longrightarrow e_2' \qquad e_3 \longrightarrow e_3'}{(if\ e_1\ e_2\ e_3) \longrightarrow (if\ e_1'\ e_2'\ e_3')}$$

Thus, an *if*-expression can "mix-evaluate" to another *if*-expression if their corresponding arguments "mix-evaluate." Notice again that the above inference rule can be written as a Horn clause.

The kinds of inference rules we have presented so far (those equivalent to Horn clauses) do not provide a natural setting for dealing with the mixed evaluation of expressions, such as $(lamb\ M)$, that contain abstractions. To handle such expressions we consider two additional meta-logical constants and the natural deduction inference rules for introducing them. Implication, written as $\Rightarrow$, is a constant of type $o \rightarrow o \rightarrow o$. Also, for every type τ built up exclusively from tm and $\rightarrow$, universal quantification at type τ will be denoted by the constant $\forall_\tau$ of type $(\tau \rightarrow o) \rightarrow o$. Quantification will be written as $\forall_\tau(\lambda x\ A)$ or more simply as $\forall_\tau x\ A$. Furthermore, we shall generally drop the type subscript when its value can be determined from context.

To prove propositions using these two new connectives, we introduce the following two introduction rules given by Gentzen [4] and Prawitz [20].

$$\frac{\begin{array}{c}(A_1)\\ \vdots \\ A_2\end{array}}{A_1 \Rightarrow A_2}\ (\Rightarrow I) \qquad\qquad \frac{A[x \mapsto c]}{\forall_\tau x\ A}\ (\forall I)$$

Here, c is a constant of type τ that must not occur in the formula $\forall x\, A$ or in any undischarged assumptions of this rule. Here, of course, $[x \mapsto c]$ denotes the operation of substituting c for free occurrences of x. Given our convention regarding non-β-normal formulas from Section 3, this rule could be simply written as

$$\frac{(Bc)}{\forall_\tau B} \quad (\forall I)$$

Here, B would be an abstraction of the form $\tau \rightarrow o$.

Both of these inference rules provide a kind of hypothetical or scoping construction in proofs. Implication introduction allows a new formula to be assumed and discharged while universal introduction allows a new constant to be introduced and discharged. Both an assumed formula and an introduced constant have very specific scopes.

Given these extensions to our proof system, we claim that the following inference rule specifies a natural mixed evaluation strategy for object-level λ-abstractions.

$$\frac{\forall x \forall y\ (x \longrightarrow y\ \Rightarrow\ ((M\, x) \longrightarrow (M'\, y)))}{(lamb\ M) \longrightarrow (lamb\ M')}.$$

To understand this rule, let us examine a simplier rule. Consider an inference rule whose premise is of the form

$$\forall x (A_1\ \Rightarrow\ A_2).$$

To construct a proof of this formula we first select a new constant, c, not occurring in A_1, A_2, or any undischarged hypothesis, and substitute it for the bound variable x. Then we assume the formula $A_1[x \mapsto c]$. This assumption will generally denote some property about this newly introduced constant. Finally, from this new assumption, we attempt to prove the formula $A_2[x \mapsto c]$. If a proof can be found, then we discharge the assumption $A_1[x \mapsto c]$ and the constant c.

Given this operational interpretation of universal and implicational propositions, the rule for the mixed evaluation of $(lamb\ M)$ can be read operationally as follows: if from the assumption that c mix-evaluates to d, where c and d are two new constants, it follows that $M\, c$ mix-evaluates to $M'\, d$, then we can conclude that $(lamb\ M)$ mix-evaluates to $(lamb\ M')$. Thus, let M be of the form $\lambda z\ t$. This inference rule can be interpreted as replacing the bound variable z in t with a new constant that will "name" that bound variable. This new name, the constant c, is also assumed to have a value, the new constant d. A value is then sought for the expression $t[z \mapsto c]$. This value, say s, will contain occurrences of d but not c. The abstraction M' is then the result of abstracting d out of s, that is, it would be the term $\lambda z\ s[d \mapsto z]$.

Figure 4 contains all the inference rules that are needed to extend the proof system for standard evaluation into the proof system for mixed evaluation. We refer to this extended system and proofs constructed in this system as the *mix* proof system and *mix*-proofs, respectively.

Note that since we have constructed this proof system by augmenting the one for standard evaluation, $\vdash_{\mathsf{se}} (e \longrightarrow \alpha)$ implies $\vdash_{\mathsf{mix}} (e \longrightarrow \alpha)$, where $\vdash_{\mathsf{mix}}$ denotes provability in the extended proof system. The converse is not true, however, and for a given e there may now be many e' such that $\vdash_{\mathsf{mix}} e \longrightarrow e'$. Also notice that mixed evaluation is *reflexive*, that is, for all terms $e \in E$, $\vdash_{\mathsf{mix}} e \longrightarrow e$.

The similarity of the rules in Figure 4 suggest that they can be explained or generated via some uniform technique. This is, indeed, the case. Before presenting such a transformation, we note that there is an alternative presentation of inference figures.

$$\frac{e_1 \longrightarrow e_1' \qquad e_2 \longrightarrow e_2' \qquad e_3 \longrightarrow e_3'}{(\mathit{if}\ e_1\ e_2\ e_3) \longrightarrow (\mathit{if}\ e_1'\ e_2'\ e_3')} \tag{2c}$$

$$\frac{\forall x \forall y\ (x \longrightarrow y\ \Rightarrow\ ((M\ x) \longrightarrow (M'\ y)))}{(\mathit{lamb}\ M) \longrightarrow (\mathit{lamb}\ M')} \qquad \frac{e_1 \longrightarrow e_1' \qquad e_2 \longrightarrow e_2'}{(e_1 @ e_2) \longrightarrow (e_1' @ e_2')} \tag{3a, 4a}$$

$$\frac{e_2 \longrightarrow e_2' \qquad \forall x \forall y\ (x \longrightarrow y\ \Rightarrow\ ((M\ x) \longrightarrow (M'\ y)))}{(\mathit{let}\ M\ e_2) \longrightarrow (\mathit{let}\ M'\ e_2')} \tag{5a}$$

$$\frac{\forall x \forall y\ (x \longrightarrow y\ \Rightarrow\ ((M\ x) \longrightarrow (M'\ y)))}{(\mathit{fix}\ M) \longrightarrow (\mathit{fix}\ M')} \tag{6a}$$

FIGURE 4
Inference Rules for the Mixed Evaluator for E

As was mentioned in Section 3, the inference rules in that section could be identified naturally with Horn clauses of the meta logic. The inference rules in this section can also be naturally identified with formulas, but they will not generally be Horn clauses. For example, the inference figure above for object-level λ-abstractions can be written as the formula

$$\forall M \forall M' [\forall x \forall y (x \longrightarrow y\ \Rightarrow\ (Mx) \longrightarrow (M'y))\ \Rightarrow\ (\mathit{lamb}\ M) \longrightarrow (\mathit{lamb}\ M')],$$

which is not a Horn clause because of the implication and universal quantifier in the antecedent. The class of formulas that is necessary here to capture the inference rules described in this section can be described as follows. Let A be a syntactic variable denoting atomic formulas of the meta-logic, that is, formulas of the form $e \longrightarrow e'$. The required class of formulas is then defined as the range of the syntactic variable L that is defined as

$$L ::= A \mid L_1 \wedge L_2 \mid L_1 \Rightarrow L_2 \mid \forall_\tau x\ L.$$

Such formulas form a subset of the hereditary Harrop formulas investigated in [14] where it is shown that in an intuitionistic proof system, these formulas can give rise to an operational interpretation similar to Horn clauses: logic programming can be naturally interpreted in hereditary Harrop formulas. The class of L-formulas is very similar to the language used for the specification of logics in the Isabelle theorem prover [17].

Given this relation between inference rules and formulas, we shall describe a syntactic transformation on constants of the abstract syntax of E (i.e., those from Figure 1) that will yield the L-formulas that encode the inference rules of Figure 4. Let t and s be terms of E that are both of type τ. The following two clauses define by recursion on simple types the three place function $[\![t \longrightarrow s : \tau]\!]$, which returns L-formulas.

- $[\![t \longrightarrow s : \tau \rightarrow \sigma]\!] := \forall x \forall y ([\![x \longrightarrow y : \tau]\!] \Rightarrow [\![tx \longrightarrow sy : \sigma]\!])$.

- $[\![t \longrightarrow s : tm]\!] := t \longrightarrow s.$

The inference rules in Figure 4 are exactly those rules that translate into the formulas denoted by $[\![c \longrightarrow c : \tau]\!]$ for each constant c (of type τ) of E's abstract syntax. These rules for mixed evaluation are therefore derived independently of standard evaluation: they only depend on the constants of the abstract syntax of E. If E were enriched with new language features then the abstract syntax would be extended with new constants. The above translation, applied to these constants, would yield an appropriate collection of mixed evaluation clauses for these new features.

Now let us consider an example of mixed evaluation. Let A be an abbreviation of the append function given by the term

$$(\mathit{fix}\ \lambda f(\mathit{lamb}\ \lambda x(\mathit{lamb}\ \lambda y(\mathit{if}\ (\mathit{null}\ x)\ y\ (\mathit{cons}\ (\mathit{car}\ x)\ (f\ (\mathit{cdr}\ x)\ y)))))).$$

Now suppose we try to show that there exists some α such that $\vdash_{se} (A\ @\ (\mathit{cons}\ 1\ \mathit{nil}))\longrightarrow\alpha$. It is not hard to see that the only possible value for α is

$$(\mathit{lamb}\ \lambda y(\mathit{if}\ (\mathit{null}\ (\mathit{cons}\ 1\ \mathit{nil}))\ y\ (\mathit{cons}\ (\mathit{car}\ (\mathit{cons}\ 1\ \mathit{nil}))\ (A\ (\mathit{cdr}\ (\mathit{cons}\ 1\ \mathit{nil}))\ y)))).$$

No further evaluation is possible.

Now consider showing $\vdash_{mix} (A\ @\ (\mathit{cons}\ 1\ \mathit{nil}))\longrightarrow\alpha$ for some α. The additional rules of the *mix* proof system provide for further simplification of this expression. In particular, the partial instantiation of a list structure often provides enough information for the evaluation of some functions, e.g., the function *null* applied to the "cons" of *any* two expressions that have values is always false. Clearly we can have the same value for α as above. Further evaluation, however, is also possible, yielding $(\mathit{lamb}\ \lambda y(\mathit{cons}\ 1\ y))$.

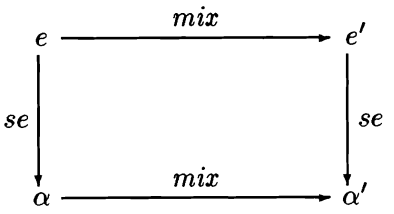
In this section we have concentrated exclusively on the declarative aspects of mixed evaluation. Of course, for an actual implementation of a mixed evaluator, the control aspects must also be addressed. Obviously, control of this evaluator is much more complex than for the standard evaluator since Proposition 3.1 does not hold for the extended proof system. For the rest of this paper, we shall assume that the mixed evaluator is a kind of non-deterministic program. The issue of imposing particular search strategies on it is beyond the scope of this paper.

5 Correctness of Mixed Evaluation

We constructed a proof system for mixed evaluation by extending the one for the standard evaluation of E. Given this intimate connection between the two systems we are able to express and prove a form of correctness for mixed evaluation directly in terms of standard evaluation. We want the mix system to preserve the values of expressions given by the standard evaluator. This notion of correctness is stated by the following theorem.

THEOREM 5.1 (Partial Correctness of Mixed Evaluation) For all $e, e', \alpha \in E$, if $\vdash_{mix} e\longrightarrow e'$ and $\vdash_{se} e\longrightarrow\alpha$ then there exists some value α' such that $\vdash_{se} e'\longrightarrow\alpha'$ and $\vdash_{mix} \alpha\longrightarrow\alpha'$.

Graphically, this relation among terms is depicted by the commuting diagram



in which the arrows $\xrightarrow{se}$ and $\xrightarrow{mix}$ correspond to provability in $\vdash_{se}$ and $\vdash_{mix}$, respectively.

The following two lemmas will help us in proving Theorem 5.1.

LEMMA 5.2 For all constants $c \in E$, $\vdash_{mix} c \longrightarrow e$ implies $e = c$. For all terms $(lamb\ M) \in E$, $\vdash_{mix} (lamb\ M) \longrightarrow e$ implies $e = (lamb\ M')$ for some M'.

The lemma provides information about the structure of *mix*-evaluated constants and abstraction. The proof trivially follows from the definition of the *mix* proof system. The next result that we need concerns the existence of *mix*-proofs for certain terms.

LEMMA 5.3 If $\vdash_{mix} t \longrightarrow s$ and $\vdash_{mix} (lamb\ M) \longrightarrow (lamb\ N)$, then $\vdash_{mix} (M\ t) \longrightarrow (N\ s)$.

We provide here only an outline of the proof. As discussed previously, the inference figures used to define *mix* can be written as a set of hereditary Harrop formulas. Let *MIX* be the set of such clauses denoting the inference rules for the *mix*-proof system. The proofs in the *mix*-system are easily identified with uniform proofs involving the formulas in *MIX* (see [14] for a definition of uniform proof). In [14] it was shown that, with respect to hereditary Harrop formulas, intuitionistic provability is the same as provability with only uniform proofs. Thus, assume $\vdash_{mix} t \longrightarrow s$ and $\vdash_{mix} (lamb\ M) \longrightarrow (lamb\ N)$. Then both of these formulas are intuitionisticaly provable from *MIX*. A uniform proof of $(lamb\ M) \longrightarrow (lamb\ N)$ is either an axiom (i.e., M is identical to N) or is constructed by first proving $\forall x \forall y (x \longrightarrow y \Rightarrow (M\ x) \longrightarrow (N\ y))$ (that is, these are the only two ways to prove such a *mix*-proposition between *lamb* expressions). Notice, that if M is identical to N, then the latter proposition is also provable. Hence, in either case, we can conclude (by universal instantiation and modus ponens) that $(M\ t) \longrightarrow (N\ s)$ has an intuitionistic proof from *MIX*. By the result in [14], it follows that $(M\ t) \longrightarrow (N\ s)$ has a uniform proof from *MIX* and this is equivalent to the fact that this same proposition has a *mix*-proof.

PROOF. (of Theorem 5.1) We assume $\vdash_{mix} e \longrightarrow e'$ and $\vdash_{se} e \longrightarrow \alpha$ for $e, e', \alpha \in E$. The proof proceeds by induction on the height h of the *se*-proof Ξ of $e \longrightarrow \alpha$ and then by case analysis on the last inference rule for a *mix*-proof Θ of $e \longrightarrow e'$. At each point, we can build the unique value α' such that $\vdash_{se} e' \longrightarrow \alpha'$ and $\vdash_{mix} \alpha \longrightarrow \alpha'$.

base: $h = 1$. Two cases apply: either e is some base constant c or of the form $(lamb\ M)$.

(i) Assume Ξ is of the form $c \longrightarrow c$ for some constant c. Then trivially Θ must be of the form $c \longrightarrow c$ and, hence $\alpha' = c$.

(ii) Assume Ξ is of the form $(lamb\ M) \longrightarrow (lamb\ M)$ for some M. Then trivially Θ must be of the form $(lamb\ M) \longrightarrow (lamb\ M')$ for some M'; and, hence, $\alpha' = (lamb\ M')$.

step: $h > 1$. We shall just consider three cases that illustrate the salient features of the proof. The other cases following similarly.

(i) Assume that the last inference rule of Ξ is of the form

$$\frac{e_1 \longrightarrow true \qquad e_2 \longrightarrow \alpha_2}{(if\ e_1\ e_2\ e_3) \longrightarrow \alpha_2}.$$

Also assume the last rule of Θ is of the form

$$\frac{e_1 \longrightarrow e_1' \qquad e_2 \longrightarrow e_2' \qquad e_3 \longrightarrow e_3'}{(if\ e_1\ e_2\ e_3) \longrightarrow (if\ e_1'\ e_2'\ e_3')}.$$

Then we must find some α_2' such that $\vdash_{se} (if\ e_1'\ e_2'\ e_3') \longrightarrow \alpha_2'$ and $\vdash_{mix} \alpha_2 \longrightarrow \alpha_2'$. By inductive hypothesis, there exists some α_1' such that $\vdash_{se} e_1' \longrightarrow \alpha_1'$ and also $\vdash_{mix} true \longrightarrow \alpha_1'$, and by Lemma 5.2, $\alpha_1' = true$; also by inductive hypothesis there exists some α_2' such that $\vdash_{se} e_2' \longrightarrow \alpha_2'$ and $\vdash_{mix} \alpha_2 \longrightarrow \alpha_2'$. These relationships among terms are illustrated by the two commuting diagrams:

$$\begin{array}{ccc} e_1 & \xrightarrow{mix} & e_1' \\ {\scriptstyle se}\downarrow & & \downarrow{\scriptstyle se} \\ true & \xrightarrow{mix} & true \end{array} \qquad \begin{array}{ccc} e_2 & \xrightarrow{mix} & e_2' \\ {\scriptstyle se}\downarrow & & \downarrow{\scriptstyle se} \\ \alpha_2 & \xrightarrow{mix} & \alpha_2' \end{array}$$

But then we have $\vdash_{se} e_1' \longrightarrow true$ and $\vdash_{se} e_2' \longrightarrow \alpha_2'$ and so we have $\vdash_{se} (if\ e_1'\ e_2'\ e_3') \longrightarrow \alpha_2'$. This is summarized by the following commuting diagram:

$$\begin{array}{ccc} (if\ e_1\ e_2\ e_3) & \xrightarrow{mix} & (if\ e_1'\ e_2'\ e_3') \\ {\scriptstyle se}\downarrow & & \downarrow{\scriptstyle se} \\ \alpha_2 & \xrightarrow{mix} & \alpha_2' \end{array}$$

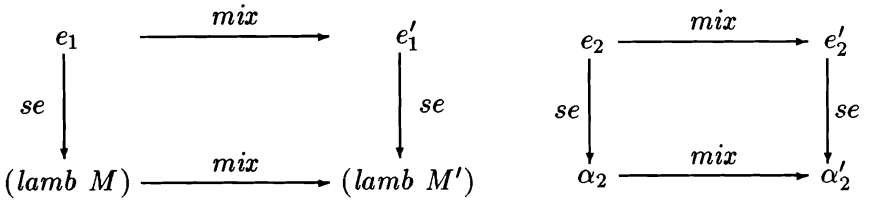
(ii) Assume the last rule of Ξ is of the form

$$\frac{e_1 \longrightarrow (lamb\ M) \qquad e_2 \longrightarrow \alpha_2 \qquad (M\alpha_2) \longrightarrow \alpha}{(e_1 @ e_2) \longrightarrow \alpha}.$$

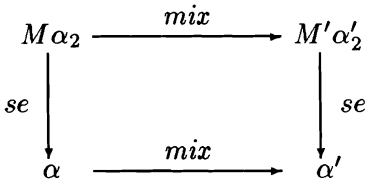
Also assume that the last inference rule of Θ is of the form

$$\frac{e_1 \longrightarrow e_1' \qquad e_2 \longrightarrow e_2'}{(e_1 @ e_2) \longrightarrow (e_1' @ e_2')}$$

By hypothesis, there exists some α_1' such that $\vdash_{se} e_1' \longrightarrow \alpha_1'$ and $\vdash_{mix} (lamb\ M) \longrightarrow \alpha_1'$, but then by Lemma 5.2, we must have $\alpha_1' = (lamb\ M')$ for some M'. (Note that we may have $M = M'$.) Likewise, there exists some α_2' such that $\vdash_{se} e_2' \longrightarrow \alpha_2'$ and $\vdash_{mix} \alpha_2 \longrightarrow \alpha_2'$. These relationships among terms are illustrated by the two commuting diagrams:



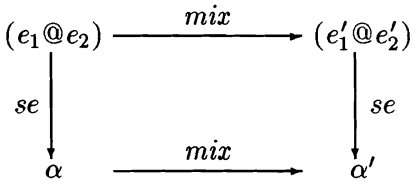
Now given $\vdash_{\text{mix}} (lamb\ \ M) \longrightarrow (lamb\ \ M')$, then either $M = M'$ or $\vdash_{\text{mix}} \forall x \forall y\ (x \longrightarrow y \ \Rightarrow\ ((M\,x) \longrightarrow (M'\,y)))$. The latter case subsumes the former so by Lemma 5.3 we have $\vdash_{\text{mix}} (M\alpha_2) \longrightarrow (M'\alpha_2')$ (since we assumed $\vdash_{\text{mix}} \alpha_2 \longrightarrow \alpha_2'$). Then, by inductive hypothesis, there exists some α' such that the following commutes:



Thus we can construct an se-proof whose last inference rule is

$$\frac{e_1' \longrightarrow (lamb\ M') \qquad e_2' \longrightarrow \alpha_2' \qquad (M'\alpha_2') \longrightarrow \alpha'}{(e_1' @ e_2') \longrightarrow \alpha'}.$$

And so the following diagram commutes:



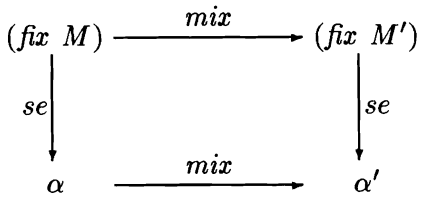
(*iii*) Assume the last rule of Ξ is of the form

$$\frac{(M\,(fix\ M)) \longrightarrow \alpha}{(fix\ M) \longrightarrow \alpha}.$$

Also assume that the last inference rule of Θ is of the form

$$\frac{\forall x \forall y\ (x \longrightarrow y \ \Rightarrow\ ((M\,x) \longrightarrow (M'\,y)))}{(fix\ M) \longrightarrow (fix\ M')}.$$

By Lemma 5.3 we have $\vdash_{\text{mix}} (M\,(fix\ M)) \longrightarrow (M'\,(fix\ M'))$ (since we assumed $\vdash_{\text{mix}} (fix\ M) \longrightarrow (fix\ M')$). By induction hypothesis, given $\vdash_{\text{se}} (M\,(fix\ M)) \longrightarrow \alpha$ and $\vdash_{\text{mix}} (M\,(fix\ M)) \longrightarrow (M'\,(fix\ M'))$, we have $\vdash_{\text{se}} (M'\,(fix\ M')) \longrightarrow \alpha'$ for some α' such that $\vdash_{\text{mix}} \alpha \longrightarrow \alpha'$. Hence we have the following commuting diagram:



The remaining cases follow similarly and are not included here. □

Notice, however, that Theorem 5.1 states that *mix* preserves values only in the forward direction. An analogous statement for the reverse direction does not hold, i.e., for some expressions e, e' such that $\vdash_{\text{mix}} e \longrightarrow e'$, e' may have a value while e has none. As an example, consider the expression $e = (\mathit{lamb}\ \lambda x.\mathit{true})@(\mathit{fix}\ \lambda x.x)$. It is easy to show that

$$\vdash_{\text{mix}} (\mathit{lamb}\ \lambda x.\mathit{true})@(\mathit{fix}\ \lambda x.x) \longrightarrow \mathit{true}$$

while $(\mathit{lamb}\ \lambda x.\mathit{true})@(\mathit{fix}\ \lambda x.x)$ has no value. For practical purposes, however, this deficiency can be overcome by enforcing a deterministic control strategy on the construction of *mix*-proofs that first applies the original (*se*) rules (when applicable) before applying the new rules of Figure 4.

This example illustrates the interaction of mixed and eager evaluation. The standard evaluator of Section 3 uses an eager evaluation strategy, i.e., arguments are evaluated before being used. (In a lazy evaluation strategy an argument is not evaluated until later, if ever.) In particular, in each of the following inference rules for standard evaluation

$$\frac{e_1 \longrightarrow (\mathit{lamb}\ M) \quad e_2 \longrightarrow \alpha_2 \quad (M\ \alpha_2) \longrightarrow \alpha}{(e_1 @ e_2) \longrightarrow \alpha} \qquad \frac{e_2 \longrightarrow \alpha_2 \quad (M\ \alpha_2) \longrightarrow \alpha}{(\mathit{let}\ M\ e_2) \longrightarrow \alpha} \qquad (4,5)$$

the argument e_2 is evaluated before it is used. But notice that these two rules, in the context of mixed evaluation, become more flexible. Recall the observation that our mixed evaluation was reflexive in that any expression can *mix*-evaluate to itself. Hence, the following two instances of these rules are derivable from the *mix* set of inference rules.

$$\frac{e_1 \longrightarrow (\mathit{lamb}\ M) \quad e_2 \longrightarrow e_2 \quad (M\ e_2) \longrightarrow \alpha}{(e_1 @ e_2) \longrightarrow \alpha} \qquad \frac{e_2 \longrightarrow e_2 \quad (M\ e_2) \longrightarrow \alpha}{(\mathit{let}\ M\ e_2) \longrightarrow \alpha}$$

for some given e_1, e_2, M, α. But these are effectively the inference rules that one uses (instead of the two above) for standard evaluation employing a lazy evaluation strategy, namely,

$$\frac{e_1 \longrightarrow (\mathit{lamb}\ M) \quad (M\ e_2) \longrightarrow \alpha}{(e_1 @ e_2) \longrightarrow \alpha} \qquad \frac{(M\ e_2) \longrightarrow \alpha}{(\mathit{let}\ M\ e_2) \longrightarrow \alpha} \qquad (4',5')$$

in which arguments are not evaluated first. We can conclude that our mixed evaluator includes not only "eager" evaluation but also "lazy" evaluation. We conjecture the following relationship between lazy evaluation and mixed evaluation.

CONJECTURE 5.4 Let *se'* be the proof system obtained by replacing rules 4 and 5 in Figure 2 by the rules 4' and 5' given above and let $\vdash_{se'}$ refer to provability in this new proof system. Then the following hold:

(*i*) For all $e, e', \alpha \in E$, if $\vdash_{\mathrm{mix}} e \longrightarrow e'$ and $\vdash_{se'} e \longrightarrow \alpha$ then there exists some value α' such that $\vdash_{se'} e' \longrightarrow \alpha'$ and $\vdash_{\mathrm{mix}} \alpha \longrightarrow \alpha'$.

(*ii*) For all $e, e', \alpha' \in E$, if $\vdash_{\mathrm{mix}} e \longrightarrow e'$ and $\vdash_{se'} e' \longrightarrow \alpha'$ then there exists some value α such that $\vdash_{se'} e \longrightarrow \alpha$ and $\vdash_{\mathrm{mix}} \alpha \longrightarrow \alpha'$.

In other words, if e *mix*-evaluates to e' then e has a value if and only if e' has a value (in the se' system) and these values are also connected by mixed evaluation.

The general discussion of correctness in this section has been greatly simplified by two features of our standard and mixed evaluation. First, the lack of explicit environments for manipulating bound variables in our specifications reduces the overall complexity of our proof systems and (meta) proofs about those systems. Second, the values for both our evaluation systems are a subset of the language of expressions, namely E. Thus we can manipulate these values just as regular expressions in the language. For specifications that include, for example, closures as values, such uniform treatment is not easily obtained.

6 Implementation of the Meta-Logic

In Sections 3 and 4, we claimed that techniques found in logic programming can be used to provide implementations of our evaluators. In this section, we elaborate a bit more on this claim.

One way to provide implementations of standard and mixed evaluation is to embedded them into Prolog-like systems. Such systems must, however, extend conventional Prolog [22] in at least the following directions.

- The collection of hereditary Harrop formulas (or the subset of L-formulas) must be supported. Prolog supports Horn clauses, which are a proper subset of hereditary Harrop formulas.
- First-order terms must be replaced with the more general notion of simply typed λ-terms.
- Unification of simply typed λ-terms must be supported to some degree. In particular, the equality of λ-terms must be determined up to α-conversion and head level β-contraction.
- Unification must also be modified to deal with the appearance of constants introduced to prove universally quantified formulas. In particular, any free or "logical" variables, say x, present when such a new constant, say c, is introduced must be constrained so that any term that instantiates x must not contain an occurrence of c.

The higher-order logic programming language λProlog [15] contains all of these extensions to conventional Prolog. Aspects of the formal foundations for λProlog can be found in [14]. The proof systems and examples in this paper were developed and tested using a prototype implementation of this language.

As we mentioned in Section 3, the depth-first search strategy of Prolog (and also of λProlog) is adequate for providing an implementation of a standard evaluator. Controlling mixed evaluation, however, is much more difficult. For this task there is probably not one correct notion of what *should*

be the *unique* result of mixed evaluation. More likely, different control strategies for mixed evaluation will be needed for different applications. Thus the depth-first control mechanisms of λProlog would not be particularly useful. Instead, a collection of high-level tactics and tacticals [6] could be employed to structure the search for proofs. Tactical and tactics can be implemented directly in λProlog [2] or in a secondary language, such as ML [17].

7 Summary and Related Work

We have shown how an elementary specification for mixed evaluation can be derived from a specification for standard evaluation. By encoding programs as simply typed λ-terms and specifying evaluation via inference rules akin to structural operational semantics, we demonstrated a natural method of extending standard evaluation with additional inference rules derived from the signature of E, the abstract syntax of a simple functional programming language. As this signature contains constants of second-order types, this extension led to inference rules that require hypothetical and scoping constructions. Such rules, fortunately, can be given a simple operational interpretation and can be naturally implemented using logic programming techniques.

Of course, the mixed evaluator that we derive provides only a "core" set of mixed evaluation rules. For example, our method is not rich enough to provide mixed evaluation rules that can simplify an expression like $max(max(x,1),x)$ to just $max(1,x)$ (given a typical definition for max) since this evaluation requires the additional information that max is both commutative and associative. This information is additional in the sense that a standard evaluator does not need it. The application of such auxiliary information is commonly found in compilers that perform such tasks as constant folding. Future work will attempt to capture this kind of auxiliary information.

The current paper provides only syntactic results concerning different forms of evaluation. We hope to extend this work to a semantic characterization that provides a natural connection between our standard and mixed evaluation systems. One promising approach uses logical relations [21]. These are relations defined over the type structure of simply typed λ-terms and they have been used to study both the syntax and semantics of the simply typed λ-calculus. The similarity between our construction of new inference rules (based on the types of constants) and the definition of logical relations is striking, but subtle differences between the two remain. We would, however, like to characterize evaluation in terms of logical relations and give semantic proofs for the soundness of mixed evaluation. Logical relations may well provide a convenient and powerful mechanism for characterizing and understanding mixed evaluation.

The use of natural deduction as a framework for evaluating programs has been studied by several others [1, 12, 19] and has been called structural operational semantics [19] and natural semantics [12]. More specifically, the discussion of mixed evaluation in the context of natural deduction or inference rules is also found in [9]. However, in that work a language independent philosophy is taken and mixed evaluation is an operation over proof trees, using a pruning-like method. A set of heuristics is developed to guide the manipulation of these proof trees. This approach is more general than ours in that it is for a specific meta-language (TYPOL) rather than for a specific object-language. Our approach, however, attempts to derive mixed evaluation automatically.

The idea of deriving mixed evaluation as an enrichment of standard evaluation was suggested by

Heering with what he calls "automatic partial ω-enrichment" [10]. This technique attempts to extend mechanically standard evaluation to mixed evaluation via a set of *enrichment rules*. Our approach appears to be an instance of these kinds of enrichment rules.

Finally, our work shares much in spirit with the automatic binding time analysis of [16]. In that work the authors present a two-level λ-calculus for distinguishing between compile-time and run-time computations. Constructions such as application and abstraction (of λ-terms) are annotated with binding information, identifying them as occurring either early (at compile-time) or late (at run-time). They describe an algorithm which, given partial binding information, computes the "best" complete binding information. The notion of best refers roughly to performing as much computation as possible at compile-time. Their ability to identify and perform computations at compile-time is similar to our ability to perform computations over terms containing symbolic values.

Acknowledgements: The first author is supported in part by a fellowship from the Corporate Research and Architecture Group, Digital Equipment Corporation, Maynard, MA USA. Both authors are supported in part by grants NSF CCR-87-05596, ONR N00014-88-K-0633, and DARPA N00014-85-K-0018.

References

[1] R. Burstall and Furio Honsell. A natural deduction treatment of operational semantics. In *Foundations of Software Technology and Theoretical Computer Science*, pages 250–269, Springer-Verlag LNCS, Vol. 338, 1988.

[2] A. Felty and D. Miller. Specifying theorem provers in a higher-order logic programming language. In *Proceedings of the Ninth International Conference on Automated Deduction*, 1988.

[3] Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Computer, Systems, Controls*, 2(5):45–50, 1971.

[4] G. Gentzen. Investigations into logical deduction. In M. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland Publishing Co., 1969.

[5] F. Giannotti, A. Matteucci, D. Pedreschi, and F. Turini. Symbolic evaluation with structural recursive symbolic constants. *Science of Computer Programming*, 9(2):161–177, 1987.

[6] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation.* Volume 78 of *Lecture Notes in Computer Science*, Springer-Verlag, 1979.

[7] J. Hannan and D. Miller. A meta-logic for functional programming. In M. Rogers and H. Abramson, editors, *Proceedings of the META88 Workshop*, MIT Press, 1989. (to appear).

[8] J. Hannan and D. Miller. Uses of higher-order unification for implementing program transformers. In K. Bowen and R. Kowalski, editors, *Fifth International Conference and Symposium on Logic Programming*, MIT Press, 1988. Also available as University of Pennsylvania Technical Report MS-CIS-88-46.

[9] L. Hascoët. Partial evaluation with inference rules. *New Generation Computing*, 6(2–3):187–209, 1988.

[10] J. Heering. Partial evaluation and ω-completeness of algebraic specifications. *Theoretical Computer Science*, 43:149–167, 1986.

[11] G. Huet and B. Lang. Proving and applying program transformations expressed with second-order logic. *Acta Informatica*, 11:31–55, 1978.

[12] G. Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, pages 22–39, Springer-Verlag LNCS, Vol. 247, 1987.

[13] D. Miller and G. Nadathur. A logic programming approach to manipulating formulas and programs. In *Proceedings of the IEEE Fourth Symposium on Logic Programming*, IEEE Press, 1987.

[14] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. To appear in the *Annals of Pure and Applied Logic.*

[15] G. Nadathur and D. Miller. An overview of λProlog. In K. Bowen and R. Kowalski, editors, *Fifth International Conference and Symposium on Logic Programming*, MIT Press, 1988.

[16] H. Nielson and F. Nielson. Automatic binding time analysis for a typed λ-calculus. *Science of Computer Programming*, 10(2):139–176, 1988.

[17] L. Paulson. The foundation of a generic theorem prover. (To appear in the *Journal of Automated Reasoning*).

[18] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, 1988.

[19] G. Plotkin. *A Structural Approach to Operational Semantics.* DAIMI FN-19, Aarhus University, Aarhus, Denmark, September 1981.

[20] Dag Prawitz. *Natural Deduction.* Almqvist & Wiksell, Uppsala, 1965.

[21] R. Statman. Logical relations and the typed λ-calculus. *Information and Control*, 65:85–97, 1985.

[22] L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques.* MIT Press, Cambridge, MA, 1986.